An Action-Centric Ontology for Software Development

Remo Dentato
Rootnode Distillery
rdentato@rootnodedistillery.eu

8 December 2025

Abstract

Software development remains plagued by conceptual fragmentation: requirements engineering, architectural design, implementation, and testing are treated as distinct disciplines with incompatible ontologies. This paper proposes a unified philosophical framework grounded in *action ontology*, the thesis that actions, not objects or states, constitute the primary reality of software systems. Drawing on Turchin's cybernetic ontology and Heylighen's theory of formality, we develop a comprehensive world view in which: (1) states are defined by the actions they enable; (2) software artifacts at all levels are decisions about action possibilities; (3) the development process is a progressive formalization along a context-dependence continuum; and (4) validation is the selection of decisions by observable reality. We demonstrate how this framework unifies the traditionally separate activities of requirements specification, architectural design, use case definition, coding, and testing under a single coherent ontology. The practical implications include novel approaches to traceability, change management, and tool design for software development environments.

Keywords: Software ontology; Action theory; Requirements engineering; Software architecture; Metasystem transitions; Formality continuum; Cybernetic epistemology

Contents

1 Introduction **3**
1.1 Philosophical Foundations . . . 3
1.2 Contributions . . . 3
1.3 Paper Structure . . . 4

2 An Action Ontology for Software **4**
2.1 The Primacy of Action . . . 4
2.2 States as Action Possibilities . . . 4
2.3 Objects as Constructed Invariants . . . 5
2.4 The Hierarchy of Abstraction . . . 5
2.4.1 The Hierarchy as Navigation Map . . . 6
2.5 Models, Decisions, and the Ground Model Problem . . . 7
2.5.1 The Evolutionary Dynamics of Models . . . 7
2.5.2 The Ground Model Problem . . . 8
2.5.3 Evolutionary Epistemology and Software Knowledge . . . 9

3 The Formality Continuum **10**
3.1 Formality as Context-Invariance . . . 10
3.2 The Impossibility of Complete Formalization . . . 10
3.3 Formality and Evolutionary Selection . . . 10
3.4 Advantages of Formality and Informality . . . 10
3.5 Optimal Formality is Context-Dependent . . . 11

4 Software Development Activities as Decision-Making **11**
4.1 Requirements Engineering . . . 12
4.2 Use Case Definition . . . 12
4.3 Architectural Design . . . 12
4.4 Implementation (Coding) . . . 13
4.5 Testing . . . 13
4.6 Deployment and Operations . . . 13
4.7 Change Management . . . 14
4.8 Debugging . . . 14

5 Validation as Selection by Reality **14**
5.1 The Bidirectional Flow . . . 14
5.2 Selection by Observable Reality . . . 15
5.3 Metasystem Transitions . . . 15
5.3.1 The Development Lifecycle as Metasystem Transition . . . 15
5.3.2 Refactoring as Metasystem Transition . . . 16

6 Practical Implications **16**
6.1 For Requirements Management . . . 16
6.2 For Software Architecture . . . 16
6.3 For Testing . . . 16
6.4 For Tool Design . . . 17
6.5 The Framework as Evaluative Lens . . . 17
6.5.1 Examples of Practices that Cohere with the Framework . . . 17
6.5.2 Examples of Practices in Tension with the Framework . . . 18
6.5.3 Applying the Evaluative Lens . . . 18

7 Related Work **19**
7.1 Process Algebra and Formal Action Models . . . 19
7.2 Foundational Software Engineering Ontologies . . . 20
7.3 Methodological and Requirements Approaches . . . 20

8 Conclusion **21**

1 Introduction

The software development industry has long sought a unified conceptual framework that could integrate its diverse activities, from early requirements elicitation through architectural design, implementation, testing, deployment, and maintenance. Despite decades of methodological innovation, from structured analysis to agile methods, we lack a coherent *ontology* that explains what software artifacts *are* and how they relate across the development lifecycle, and an *epistemology* that accounts for how we build, validate, and evolve our knowledge of the system being developed.

This fragmentation manifests in practical problems: requirements that cannot be traced to implementations, architectural decisions that conflict with business needs, tests that verify compliance with specifications but not with user intent, and pervasive miscommunication between stakeholders who operate with incompatible mental models of the same system.

We propose that these problems stem from a fundamental ontological error: treating software systems as collections of *objects* (entities, components, modules) rather than as patterns of *action*. This paper develops an action-centric ontology for software development, grounded in rigorous philosophical foundations, that unifies all development activities under a single coherent framework. Complementing this ontology, we propose an evolutionary epistemology that explains how stakeholders build, validate, and refine their knowledge of the system through iterative cycles of decision-making and selection by reality.

1.1 Philosophical Foundations

Our framework synthesizes four philosophical streams:

(i) **Turchin's Cybernetic Ontology of Action** [Turchin, 1993]: The thesis that action is ontologically primary, with states defined as sets of possible actions, and objects constructed as invariants under cognitive transformations.

(ii) **Heylighen's Theory of Formality** [Heylighen, 1999]: The analysis of formality as context-invariance, with the recognition that complete formalization is impossible and that optimal formality is context-dependent.

(iii) **Evolutionary Constructivism** [Heylighen, 2000]: The view that knowledge systems evolve through variation and selection, with metasystem transitions creating new levels of control and organization.

(iv) **Process-Theoretic Foundations**: Our ontology aligns with, but is distinct from, the process-algebraic tradition in computer science (Hoare's CSP [Hoare, 1985], Milner's CCS and π-calculus [Milner, 1989, 1999]) which treats processes rather than data structures as computational primitives. While process algebras provide *syntactic calculi* for compositional reasoning, our framework addresses the *epistemological* question of how knowledge of systems is constructed through action.

1.2 Contributions

This paper makes the following contributions:

1. A formal ontology for software development grounded in action theory
2. A unified model of software artifacts as decisions about action possibilities
3. An account of the development lifecycle as progressive formalization
4. A theory of validation as selection by observable reality
5. Practical implications for requirements management and development tools

1.3 Paper Structure

Section 2 presents the foundational action ontology. Section 3 develops the formality continuum for software artifacts. Section 4 applies the framework to traditional development activities. Section 5 addresses validation and evolution. Section 6 discusses practical implications. Section 7 reviews related work. Section 8 concludes.

2 An Action Ontology for Software

2.1 The Primacy of Action

Following Turchin [Turchin, 1993], we adopt the fundamental ontological thesis:

Principle 1 (Primacy of Action)**.** *The primary reality of software systems consists of actions, not objects or states. Objects and states are representations constructed from patterns of action.*

This principle inverts the conventional view. We typically imagine software as a collection of objects (users, accounts, transactions) that exist in states and occasionally undergo changes. The action-centric view holds that only the interactions, the actions, are real; objects and states are mental constructs we create to organize our understanding of action patterns.

Definition 1 (Action)**.** *An **action** is an elementary event that constitutes an interaction between agents (which may include users, system components, or external systems). Actions are characterized by:*

- *An agent $\mathcal{G}$ that performs the action*
- *A domain $\mathcal{A}$ of possible actions*
- *An observable effect on subsequent action possibilities*

Note that "elementary" is relative to the level of abstraction at which the agent operates. An action that appears atomic at Level 2 (Perceptions) may decompose into multiple Level 1 events (Sensations) and countless Level 0 operations. We do not require ACID atomicity; an action is elementary if the agent treats it as a single indivisible interaction for the purposes of their model. This relativity of atomicity echoes Lamport's [Lamport, 1978] observation that in distributed systems, simultaneity and ordering are observer-relative—what appears as a single event from one vantage point may decompose into a sequence from another.

2.2 States as Action Possibilities

The radical move in Turchin's ontology is defining states in terms of actions:

Definition 2 (State)**.** *A **state** $s \in \mathcal{S}$ of a system is the set of actions possible in that state, together with their associated probabilities:*

$$s = \{(a, p_a) : a \in \mathcal{A}_s, p_a \in [0, 1]\}$$

Two states in which exactly the same actions are possible with the same probabilities are identical, as there is no way to distinguish them.

This definition has profound implications for software specification:

Corollary 1. *Specifying a system state is equivalent to specifying which actions are enabled or prevented in that state. A requirement that does not ultimately constrain action possibilities is vacuous.*

2.3 Objects as Constructed Invariants

If states are action possibilities, what are objects? Turchin provides a sophisticated answer:

Definition 3 (Object)**.** *An **object** b is a family of transformations $\{f_a : a \in \mathcal{A}_{cog}\}$ that predicts how representations change under cognitive actions, actions whose purpose is to acquire knowledge about the system rather than to change it:*

$$b = \{f_a : R_{ob} \to R_{ob} \mid a \in \mathcal{A}_{cog}\}$$

where R_{ob} is the set of representations associated with the object.

An object is thus not a thing "out there" but a pattern of stability that we construct. We perceive "a user account" as an object because our representations remain stable (in predictable ways) as we perform various cognitive actions: logging in from different devices, viewing as admin versus user, accessing across sessions.

Proposition 1 (Object Relativity)**.** *Different agents may construct different objects from the same underlying action patterns, depending on which cognitive actions they perform.*

This explains a persistent problem in software development: stakeholders disagree about domain models not because one is "right" and others "wrong," but because they perform different cognitive actions and thus construct different invariants.

2.4 The Hierarchy of Abstraction

Turchin identifies a hierarchical structure in knowledge systems:

1. **Level 0: Actions**, elementary computational events (CPU cycles, memory operations, network packets)
2. **Level 1: Sensations**, observable interactions (clicks, responses, displayed content, errors, log entries)
3. **Level 2: Perceptions**, constructed objects and mental models ("the checkout flow," "my account," "the dashboard")
4. **Level 3: Conceptions**, formal expressions (requirements documents, API specifications, code, test cases)

Figure 1 illustrates this hierarchy and its correspondence to software development artifacts.

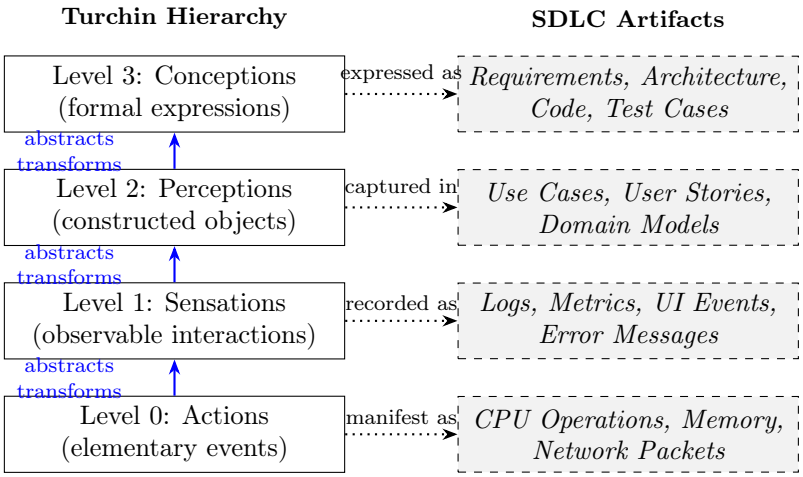

Figure 1: The Turchin hierarchy of abstraction and its correspondence to software development artifacts. Each level abstracts *transformations* of the level below. Arrows indicate the direction of abstraction; dotted lines show how each cognitive level manifests in development practice.

Principle 2 (Hierarchical Abstraction)**.** *Higher levels of abstraction represent transformations of lower-level representations, not the representations themselves. Each level is related to the level below by a metasystem transition.*

2.4.1 The Hierarchy as Navigation Map

Beyond its theoretical significance, the abstraction hierarchy serves as a *navigation map* for locating problems, preparing communications, and validating refinements.

Diagnosing Model Misalignment. When stakeholders disagree or system behavior contradicts expectations, the hierarchy identifies *where* misalignment occurred:

- **Level 1**: Stakeholders observe different sensations (one sees an error, another does not). *Problem*: observation.
- **Level 2**: Stakeholders construct different objects from identical sensations (both see the same screen but perceive different states). *Problem*: interpretation.
- **Level 3**: Formal expressions fail to capture perceptions ("fast response" means different things to different stakeholders). *Problem*: formalization.

Locating the level directs corrective action: Level 1 problems require better instrumentation; Level 2, shared vocabulary; Level 3, precise specification.

Preparing Views for Stakeholders. Different stakeholders operate at different levels:

- **End users**: Level 2 ("the checkout flow," "my saved items")
- **Developers**: Levels 2–3 (translating perceptions to code)
- **Operations**: Level 1 (logs, metrics, error rates)
- **Business stakeholders**: Level 2, occasionally Level 3

Effective communication translates models to the level where the audience can validate them.

Tracing Decisions Through Levels. Debugging and change analysis trace from symptoms to causes:

1. Problem manifests at **Level 1**: error, unexpected output, failed test
2. Interpret at **Level 2**: which perception was violated?
3. Trace to **Level 3**: which formal decision produced this?
4. Identify root cause: wrong implementation, design, or intent?

This "forensic epistemology" prevents fixing symptoms (Level 1) without addressing causes (Levels 2–3).

Validating Refinements. Each refinement step should preserve meaning across levels:

- Does **code** (Level 3) produce **sensations** (Level 1) that stakeholders interpret as intended **perceptions** (Level 2)?
- Does **architecture** (Level 3) support the action possibilities stakeholders **perceive** (Level 2) as system capabilities?
- Do **requirements** (Level 3) capture stakeholder **perceptions** (Level 2)?

Validation fails when refinement produces correct behavior at one level but violates expectations at another.

2.5 Models, Decisions, and the Ground Model Problem

We must first distinguish *agents* (entities that perform actions) from *stakeholders* (entities that build knowledge about the system).

Definition 4 (Stakeholder)**.** *A **stakeholder** is an entity that participates in building knowledge about the system: holding models, making decisions, and contributing to artifacts that constrain agent actions.*

Proposition 2 (Agent-Stakeholder Relationship)**.** *Some entities are both agents and stakeholders; others are only one. A **user** is typically both; a **business sponsor** may influence decisions without using the system; a **system component** performs actions but does not participate in knowledge-building. Development translates stakeholder models into constraints on agent actions.*

Definition 5 (Model)**.** *A **model** is a stakeholder's cognitive representation of the system's action possibilities: which actions are possible in which states, their effects, and how they relate to goals.*

While artifacts record *decisions* about action possibilities, stakeholders hold *models*—mental representations that drive and are shaped by decisions.

Principle 3 (Model-Decision Duality)**.** *Stakeholders' models drive decisions; decisions, once recorded, influence models as understanding evolves. Neither is primary: they co-evolve through continuous interaction.*

This duality illuminates persistent challenges: different stakeholders construct different objects because they hold different models (Proposition 1); requirements elicitation is difficult because stakeholders cannot fully articulate their models; miscommunication occurs when stakeholders believe their models align but actually differ.

2.5.1 The Evolutionary Dynamics of Models

Stakeholder models are *hypotheses* subject to evolutionary selection. A model predicts: "If I perform action X in state S, the system responds with Y." When reality contradicts this prediction, the model must evolve.

Principle 4 (Model Selection)**.** *Models evolve when they fail to predict the correspondence between system actions and observable reality. Models that generate correct predictions survive; those that fail are revised or abandoned.*

This selection pressure operates universally: users refine models through interaction; developers through debugging; organizations through retrospectives.

The complete evolutionary cycle:

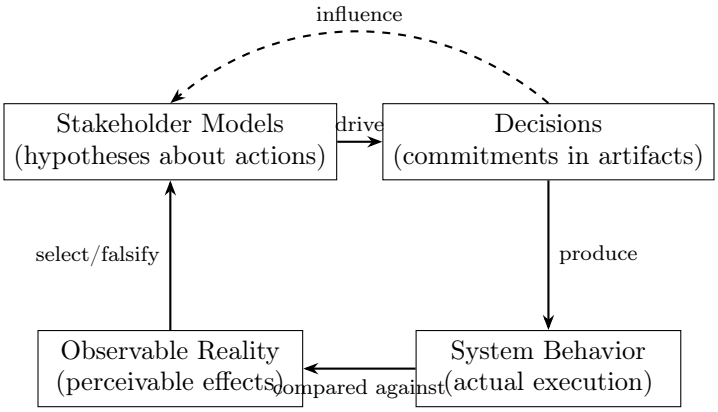

Figure 2: The evolutionary cycle of models and decisions. "System Behavior" denotes actual execution (latent, internal); "Observable Reality" denotes perceivable effects (Level 1 Sensations).

This cycle explains why software development is inherently iterative: incomplete models produce surprises when confronted with reality, driving model evolution. The goal is not to eliminate this cycle but to accelerate it through shorter feedback loops.

Corollary 2 (Documentation as Model Transmission)**.** *Documentation transmits models between stakeholders, reducing costly trial-and-error learning and accelerating model convergence.*

Corollary 3 (Bugs as Model Failures)**.** *A bug is a model failure: some stakeholder's model failed to predict actual behavior. Debugging revises models to restore predictive accuracy, then revises decisions to align the system with corrected understanding.*

2.5.2 The Ground Model Problem

The most difficult part of software development occurs at its origin—formalizing stakeholder models into explicit decisions. Following Börger's analysis of Abstract State Machines [Börger, 1999], this link between informal models and formal decisions cannot be justified purely by theoretical means.

Definition 6 (Ground Model)**.** *A **ground model** is the initial formal specification that captures informal stakeholder models at the appropriate abstraction level such that:*

(a) It can be understood and validated by domain experts

(b) It defines every feature relevant to user expectations

(c) It contains only what the problem logic requires, without premature implementation decisions

Connecting the Ground Model to Action Invariants. The ground model can be precisely characterized in action-ontology terms. Recall that objects are invariants under cognitive actions (Definition 3). The ground model is the *formalization of invariants shared across stakeholders.*

Proposition 3 (Ground Model as Shared Action Invariants)**.** *The ground model is the minimal set of action invariants that:*

(a) Are constructed identically by all relevant stakeholders (achieving model alignment at Level 2)

(b) Can be expressed formally (transitioning to Level 3)

(c) Suffice to constrain subsequent solution decisions

Invariants that differ across stakeholders, or resist formalization, remain as contextual assumptions requiring explicit documentation.

This explains why ground model construction is difficult: it requires identifying which cognitive actions stakeholders *share* versus which differ. When stakeholders perform different cognitive actions, they construct different objects from the same underlying patterns. The ground model must be grounded in invariants that survive *all* stakeholder probes.

Corollary 4 (Ground Model Validation)**.** *A ground model is valid iff the action invariants it formalizes correspond to invariants actually constructed by stakeholders. Validation requires cognitive actions (reviews, walkthroughs, prototypes) testing whether stakeholders recognize formalized invariants as matching their own constructions.*

Principle 5 (Pragmatic Foundation)**.** *Ground model justification is necessarily pragmatic, not purely theoretical:*

(a) ***Conceptual justification****: Formal constructs correspond one-to-one to action invariants constructed by stakeholders*

(b) ***Experimental justification****: Falsifiability criteria (acceptance tests) allow validation against observable reality*

The ground model occupies a critical position on the formality continuum: formal enough to guide subsequent decisions, informal enough to be validated against stakeholder models.

2.5.3 Evolutionary Epistemology and Software Knowledge

The action-centric view parallels evolutionary epistemology [Heylighen, 2000, Campbell, 1974]:

Table 1: Parallel between evolutionary epistemology and software development

Evolutionary Epistemology	**Software Development**
Knowledge as model for prediction	Specification as model for behavior
Variation through hypothesis generation	Variation through design alternatives
Selection by environmental feedback	Selection by testing and validation
Fitness as predictive adequacy	Fitness as stakeholder satisfaction
No absolute truth, only adequacy	No perfect specification, only sufficiency

Campbell's *vicarious selectors* [Campbell, 1974] illuminate software artifacts: specifications, architectures, and tests function as internal selectors that evaluate implementations before they encounter the ultimate selector, observable reality in production.

Proposition 4 (Software as Vicarious Selector)**.** *Each artifact level (requirement, design, test) functions as a vicarious selector, eliminating inadequate action possibilities before realization.*

This explains why complete specifications are impossible: software specifications cannot exhaustively capture all action possibilities but only constrain them sufficiently for the intended context.

3 The Formality Continuum

3.1 Formality as Context-Invariance

Heylighen [Heylighen, 1999] provides a rigorous definition:

Definition 7 (Formality)**.** *The **formality** of an expression is the degree to which its meaning is invariant across changes in context. An expression is maximally formal if its meaning remains identical regardless of who interprets it, when, or in what situation.*

Formality thus forms a continuum:

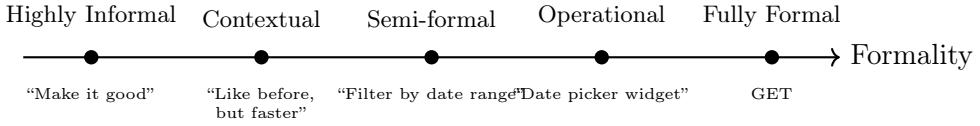

Figure 3: The formality continuum for software specifications

3.2 The Impossibility of Complete Formalization

A crucial insight from Heylighen's analysis:

Principle 6 (Incompleteness of Formalization)**.** *Every formal system contains: (a) **primitive terms** understood through shared context; (b) **background assumptions** that cannot be exhaustively enumerated; (c) **operational setups** involving apparatus with unknown properties. Complete elimination of context is impossible; Gödel's incompleteness and Heisenberg's uncertainty are specific instances.*

This has immediate practical consequences: *the quest for "complete" requirements is misguided.* The question is whether specifications are *sufficiently formal* for the decisions they must support.

3.3 Formality and Evolutionary Selection

Higher formality increases an artifact's *selectivity*—its capacity to distinguish fit from unfit implementations:

Proposition 5 (Formality-Selectivity Relationship)**.** *Low formality yields weak selection (many implementations pass, including unfit ones); high formality yields strong selection (fewer pass, but selection may be premature). Optimal formality balances selective power against eliminating viable solutions too early.*

This explains the typical progression from low to high formality: early stages benefit from weak selection (exploring possibilities), later stages require strong selection (converging on implementation). The ground model occupies a critical middle position.

3.4 Advantages of Formality and Informality

Heylighen identifies distinct advantages at each end:

Table 2: Trade-offs along the formality continuum

Advantages of Formality	**Advantages of Informality**
Storability (persists over time)	Flexibility and adaptability
Universal communicability	Intuitive understanding
Testability	Creates involvement
Accumulation of knowledge	Expresses novel meanings
Supports tool automation	Less cognitive load

3.5 Optimal Formality is Context-Dependent

Principle 7 (Context-Dependent Optimality)**.** *Optimal formality depends on: (a)* ***domain stability*** *(stable domains permit higher formality); (b)* ***stakeholder distance*** *(greater distance requires higher formality); (c)* ***decision criticality*** *(critical decisions warrant higher formality); (d)* ***stage of understanding*** *(earlier stages benefit from lower formality).*

This explains why agile methods succeed in volatile domains with co-located teams, while safety-critical development requires rigorous documentation for stable domains with distributed, regulated stakeholders.

4 Software Development Activities as Decision-Making

We now apply the framework to traditional development activities. Our thesis: all development activities are *decision-making processes* that progressively constrain action possibilities at increasing formality levels.

Developing a software system means evolving a body of decisions about action possibilities, shared among stakeholders, expressed at varying formality levels, and continuously selected by observable reality.

For each activity, we present the traditional view, then reconceptualize it through the action-centric lens.

Definition 8 (Development Decision)**.** *A* ***development decision*** d *is a tuple:*

$$d = (\mathcal{G}, \mathcal{A}^+, \mathcal{A}^-, \mathcal{F}, \mathcal{C}, \rho, \sigma)$$

where:

- $\mathcal{G}$ *is the agent (decision-maker) responsible*
- $\mathcal{A}^+$ *is the set of actions to be enabled*
- $\mathcal{A}^-$ *is the set of actions to be prevented*
- $\mathcal{F} \in [0, 1]$ *is the formality level*
- $\mathcal{C}$ *is the context required for interpretation*
- ρ *is the rationale, representable as typed edges in the decision provenance graph:* $\rho \subseteq \{(d_i, d_j, \tau) \mid d_i, d_j \in \mathcal{D}, \tau \in \{\textit{constrains}, \textit{enables}, \textit{realizes}, \textit{questions}\}\}$
- $\sigma \in \{\textit{hypothesis}, \textit{belief}, \textit{evidence}, \textit{proven}\}$ *is the confidence level*

4.1 Requirements Engineering

Traditional View. Requirements engineering discovers, documents, and validates system needs, typically expressed as statements about properties or behaviors.

Action-Centric Reconceptualization.

Definition 9 (Requirement)**.** *A **requirement** is a decision about which actions should be possible (or prevented) in which states, with sufficient confidence to constrain subsequent solution decisions.*

Implications:

1. **Action-oriented expression**: Specify actions enabled/prevented, not object properties (e.g., "Users can add/remove items, view cart, checkout" rather than "The system shall have a shopping cart")
2. **Explicit formality management**: Different stages appropriately have different formality levels
3. **Context as first-class citizen**: Document required context rather than assuming specifications are context-free

Requirements Traceability Reconsidered. Traceability becomes tracking *decision provenance*:

Definition 10 (Decision Provenance)**.** *The **provenance** of a decision d is the directed graph of decisions that constrain, enable, realize, or question d, with typed edges:*

- $d_1 \xrightarrow{constrains} d_2$*: d_1 limits options for d_2*
- $d_1 \xrightarrow{enables} d_2$*: d_1 makes d_2 possible*
- $d_1 \xrightarrow{realizes} d_2$*: d_2 implements d_1 at higher formality*
- $d_1 \xrightarrow{questions} d_2$*: Evidence from d_1 challenges d_2*

4.2 Use Case Definition

Traditional View. Use cases describe interaction sequences between actors and system to achieve goals, typically semi-formal narratives.

Action-Centric Reconceptualization.

Definition 11 (Use Case)**.** *A **use case** specifies a coherent action sequence $a_1, \ldots, a_n$ transforming initial state s_0 to goal state s_g, with alternative and exception sequences.*

Use cases occupy a specific position on the formality continuum: more formal than raw requirements, less formal than implementation. In the Turchin hierarchy, they belong to Level 2 (Perceptions): constructed invariants under stakeholder reasoning about system behavior.

4.3 Architectural Design

Traditional View. Architecture concerns high-level structure: components, connectors, and configurations.

Action-Centric Reconceptualization.

Definition 12 (Architectural Decision)**.** *An **architectural decision**: (a) partitions the action space into domains assigned to components; (b) specifies interfaces, actions possible between components; (c) establishes constraints on parallel vs. sequential actions.*

Principle 8 (Architecture as Action Partitioning)**.** *Architecture is fundamentally about organizing action possibilities. A component boundary decides which actions are internal (invisible) and which are external (interface).*

Architectural Styles as Action Patterns. Common styles have characteristic action patterns: *Layered*, layer n invokes only layer $n-1$; *Microservices*, disjoint action domains, cross-service via network; *Event-driven*, components publish events, no direct invocation; *Pipe-and-filter*, each component's outputs are the next's inputs.

4.4 Implementation (Coding)

Traditional View. Implementation translates designs into executable code.

Action-Centric Reconceptualization.

Definition 13 (Implementation)**.** ***Implementation*** *is maximally formal specification: code executable by a machine to perform specified actions.*

Code occupies the highest formality level, interpretable by a machine, the ultimate context-independent interpreter. Yet even code depends on language semantics, runtime, and hardware. Code provides an *operational definition* of actions: a procedure determining whether an action occurs.

Programming Paradigms as Action Models. Different paradigms model action composition differently: *Imperative*, sequential state-modifying actions; *Functional*, composition of pure functions; *Object-oriented*, actions organized by performing objects; *Actor*, concurrent agents responding to messages; *Reactive*, actions triggered by event streams.

4.5 Testing

Traditional View. Testing verifies software behavior against specifications.

Action-Centric Reconceptualization.

Definition 14 (Test)**.** *A* ***test*** *is a cognitive action gathering evidence about whether other actions are possible in specified states.*

This unifies all testing forms: *unit tests* verify component interface actions; *integration tests* verify cross-component actions; *system tests* verify end-user action outcomes; *acceptance tests* verify stakeholder intent satisfaction.

Principle 9 (Testing as Selection)**.** *Testing implements evolutionary selection: decisions (hypotheses about action possibilities) are tested against observable reality; failures trigger revision.*

4.6 Deployment and Operations

Traditional View. Deployment makes software available; operations manages running systems.

Action-Centric Reconceptualization.

Definition 15 (Deployment)**.** ***Deployment*** *transitions from a state where user actions are impossible to one where they become possible—the realization of action possibilities in observable reality.*

Definition 16 (Operations)**.** ***Operations*** *maintains system state such that intended user actions remain possible with acceptable probability.*

4.7 Change Management

Traditional View. Change management handles modification requests through impact analysis, approval, and controlled implementation.

Action-Centric Reconceptualization.

Definition 17 (Change Request)**.** *A **change request** revises existing action possibilities in response to information that questions decisions at some hierarchy level.*

Change requests originate when observable reality, user feedback, market shifts, defects, challenges previously accepted decisions. Change management traces impact through the decision provenance graph: which intent decisions enabled this solution? Which solution decisions constrained this implementation? Which artifacts must be revised?

Systems with clear decision provenance assess change impact systematically; those without rely on tacit knowledge and guesswork.

4.8 Debugging

Traditional View. Debugging identifies and removes defects by reproducing, isolating, understanding, and fixing problems.

Action-Centric Reconceptualization.

Definition 18 (Debugging)**.** ***Debugging** traces a discrepancy between expected and actual behavior back to its originating decision.*

Debugging follows "questions" links upward through the hierarchy: (1) problem manifests at Level 1 (error, unexpected output); (2) interpret at Level 2 (which perception violated?); (3) trace to Level 3 (which decision produced this?); (4) identify root cause (wrong implementation, design, or intent?).

Principle 10 (Debugging as Forensic Epistemology)**.** *Debugging reconstructs which hypothesis about action possibilities was falsified. The fix requires revising the falsified decision at the correct level—fixing at the wrong level causes recurrence or new problems.*

5 Validation as Selection by Reality

5.1 The Bidirectional Flow

Information flows both "downward" (intent to implementation) and "upward" (reality to intent):

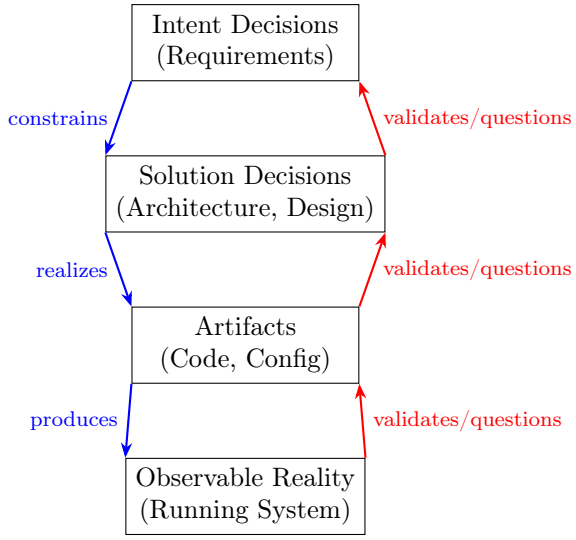

Figure 4: Bidirectional flow between decision levels

5.2 Selection by Observable Reality

Principle 11 (Reality as Selector)**.** *Observable reality, actual system behavior and its effects, is the ultimate selector of decisions at all levels. Decisions leading to systems whose actions do not match intent are invalidated and must be revised.*

Crucially, *validation is not a phase but a continuous process.* Every observation of the running system is potentially an act of selection.

5.3 Metasystem Transitions

When reality consistently questions decisions at a given level, the response may be creating a new level of organization:

Definition 19 (Metasystem Transition)**.** *A* ***metasystem transition*** *[Turchin, 1977] is the emergence of a new abstraction or control level that coordinates variety at lower levels, characterized by: (a) assembly of subsystems into a new whole; (b) emergence of higher-level control; (c) new properties absent in subsystems alone; (d) increased variety at object level with unified control at meta-level.*

Metasystem transitions are evolution's "quanta", discontinuous jumps transforming organization itself [Heylighen, 2000]. In software: *frameworks* extract common control from multiple applications; *platforms* provide unified infrastructure; *abstraction layers* encapsulate leaking details; *methodologies* control the development process itself.

Principle 12 (Progressive Control)**.** *Each metasystem transition increases freedom and adaptivity while establishing higher-level constraints. The new control level coordinates (not eliminates) lower-level variety, enabling more sophisticated behavior.*

5.3.1 The Development Lifecycle as Metasystem Transition

The lifecycle is a sequence of metasystem transitions: *Requirements* → *Ground Model* (informal needs controlled by formal specification); *Ground Model* → *Architecture* (functional requirements controlled by structural decisions); *Architecture* → *Implementation* (design controlled by executable code); *Implementation* → *Deployment* (code controlled by operational infrastructure). Each increases formality while upward validation may trigger revision at any level.

5.3.2 Refactoring as Metasystem Transition

Refactoring, restructuring code without changing external behavior, exemplifies metasystem transition: deliberately creating new control levels in response to evolutionary pressure.

Definition 20 (Refactoring)**.** ***Refactoring*** *assembles code fragments into new organizational structures providing higher-level control over previously scattered action variety.*

Not all refactorings are metasystem transitions. *Lateral* refactorings (Rename Variable, Reorder Parameters) improve clarity without new abstraction. *Vertical* refactorings [Fowler, 1999] create new control levels:

Table 3: Selected refactorings as metasystem transitions

Refactoring	**Transition**	**New Control Level**
Extract Method	Assembly of statements	Method as action coordinator
Extract Class	Assembly of methods	Class as responsibility coordinator
Extract Interface	Assembly of signatures	Interface as contract coordinator
Replace Conditional with Polymorphism	Assembly of branches	Type hierarchy as behavior coordinator

Principle 13 (Refactoring Pressure)**.** *Refactoring need arises when organizational structure fails to control action variety. Code smells signal insufficient meta-level control: Duplication—same pattern without unifying control; Long Method—too many actions without coordination; Feature Envy—actions misallocated to wrong domains.*

Without periodic metasystem transitions, systems accumulate uncontrolled variety. Technical debt is *deferred metasystem transitions*, maintaining systems whose control structures are inadequate for current variety. Refactoring can propagate upward: when restructuring reveals the ground model missed essential patterns, revise the ground model itself.

6 Practical Implications

6.1 For Requirements Management

The framework suggests: (1) *action-oriented specification*—express requirements as actions enabled/prevented; (2) *explicit formality management*—track formality levels, accept variation; (3) *context documentation*—capture required context explicitly; (4) *confidence tracking*—distinguish hypotheses from proven decisions; (5) *bidirectional traceability*—support "questions" links from reality, not just "realizes" links downward.

6.2 For Software Architecture

Describe architectures as action partitions and interfaces; define interfaces by enabled actions and pre/post-conditions; document architectural decisions as action-space constraints.

6.3 For Testing

Design tests as cognitive actions gathering evidence; measure coverage by action possibilities exercised; treat production monitoring as continuous selection.

6.4 For Tool Design

Action-centric tools would: model decisions with full structure (agent, actions, formality, context, rationale, confidence); support the formality continuum; track bidirectional relationships; integrate runtime observation; support multiple stakeholder constructions of the same domain.

6.5 The Framework as Evaluative Lens

The action-centric framework is not prescriptive: it does not mandate specific methodologies, tools, or organizational structures. Instead, it provides an *evaluative lens*, a set of criteria for assessing whether a practice coheres with the ontological and epistemological principles developed in this paper.

Principle 14 (Framework Alignment Criteria)**.** *A development practice coheres with the action-centric framework to the extent that it:*

(a) Reduces the context required for stakeholders to achieve higher formality (supporting Principle 7)

(b) Accelerates the evolutionary cycle between decisions and reality (supporting Principle 4)

(c) Improves alignment of action invariants across stakeholders (supporting Proposition 3)

(d) Makes selection pressure visible earlier through vicarious selectors (supporting Principle 9)

(e) Supports appropriate formality for the current stage of understanding (supporting Principle 6)

(f) Enables traceability through decision provenance (supporting Definition 8)

A practice is in tension with the framework to the extent that it violates these criteria. Note that tension with the framework does not necessarily imply the practice is wrong in all contexts; it implies that the practice works against the dynamics the framework identifies as central to effective software development.

This evaluative approach allows practitioners to assess any activity, organizational choice, or tool adoption without requiring adherence to a particular methodology. The framework provides *reasons* for why certain practices tend to succeed or fail, grounded in the ontological structure of software development itself. The following examples illustrate how the framework applies:

6.5.1 Examples of Practices that Cohere with the Framework

Organizing teams by business domain reduces the context needed for communication among stakeholders. When team members share domain knowledge, stakeholders can achieve higher formality faster because less context must be explicitly communicated. *Theoretical grounding*: This increases the likelihood that stakeholders construct identical action invariants (Proposition 3), reducing Object Relativity and accelerating ground model convergence.

Using diagrams to share knowledge increases the formality of representations exchanged between stakeholders. A diagram is more context-invariant than a verbal description. *Theoretical grounding*: Diagrams move communication rightward on the formality continuum (Figure 3), reducing dependence on shared context and making model misalignment visible.

Continuous integration with automated tests creates rapid feedback loops between decisions (code commits) and selection (test results). *Theoretical grounding*: This accelerates the evolutionary cycle (Figure 2) and instantiates Principle 9 by making selection pressure visible at Level 3 rather than waiting for Level 1 failures in production.

Refactoring when code smells emerge responds to signals that organizational structure inadequately controls action variety. *Theoretical grounding*: Code smells indicate violations of Principle 13; refactoring restores adequate meta-level control over action variety.

Cross-functional reviews expose decisions to multiple stakeholder models before propagation to implementation. *Theoretical grounding*: Reviews test whether formalized action invariants match diverse stakeholder constructions, validating the ground model per Corollary 4.

6.5.2 Examples of Practices in Tension with the Framework

Skipping acceptance tests removes experimental justification for the ground model. *Theoretical grounding*: This violates Principle 5, which requires experimental justification. The ground model becomes unfalsifiable, blocking the "questions" link in decision provenance.

Siloed teams with no shared vocabulary increase context required for cross-boundary communication. *Theoretical grounding*: Without shared cognitive actions, stakeholders construct different action invariants (Proposition 1). Conditions for a valid ground model (Proposition 3) cannot be met.

Documentation written after implementation attempts to transmit obsolete models. *Theoretical grounding*: This inverts Model-Decision Duality (Principle 3). Documentation should transmit models to *drive* decisions; post-hoc documentation loses the rationale (ρ) that justified them.

Ignoring production feedback blocks upward "questions" flow (Figure 4). *Theoretical grounding*: This violates Principle 11. Without selection pressure from reality, evolutionary dynamics (Principle 4) cannot operate.

Premature formalization imposes high formality before models have evolved sufficiently. *Theoretical grounding*: This violates Principle 7, criterion (d): earlier stages benefit from lower formality.

6.5.3 Applying the Evaluative Lens

When evaluating any proposed practice, practitioners can ask:

- Does this reduce or increase the context required for shared understanding? (Formality continuum)
- Does this speed up or slow down feedback from reality? (Evolutionary cycle)
- Does this help or hinder alignment of action invariants across stakeholders? (Ground model validity)
- Does this create vicarious selectors or remove them? (Testing as selection)
- Is the formality level appropriate to our current stage of understanding? (Context-dependent optimality)
- Can we trace decisions to their origins if reality questions them? (Decision provenance)

These questions transform the framework from an abstract philosophy into a practical tool for decision-making about development practices. The framework does not prescribe *what* to do; it provides *theoretical reasons* for evaluating whether what you are doing serves the fundamental goals of software development: evolving a body of decisions about action possibilities that, when realized in a system, will survive selection by observable reality.

Importantly, a practice in tension with the framework is not necessarily *wrong*, context matters. A startup exploring product-market fit may rationally skip acceptance tests because the cost of building them exceeds the benefit when the product itself may pivot. The framework does not condemn this choice; it *explains* the trade-off: by removing experimental justification, the team accepts higher risk that their ground model diverges from stakeholder needs. The framework makes such trade-offs explicit rather than leaving them as tacit assumptions.

7 Related Work

Our framework engages with several distinct research traditions. We organize this discussion around three questions: (1) How does our action ontology relate to mathematical formalizations of process and concurrency? (2) How does it position against the dominant "thing-centric" ontologies in software engineering? (3) How does it connect to existing methodological approaches?

7.1 Process Algebra and Formal Action Models

The primacy of action we advocate has a rigorous mathematical counterpart in *process algebra*, a family of formal calculi that treat processes (patterns of action) as the fundamental computational primitives.

Hoare's *Communicating Sequential Processes* (CSP) [Hoare, 1985] defines systems entirely by their *traces*: sequences of observable events. A process in CSP has no intrinsic "state" apart from the events it can engage in and the events it has performed. This mirrors our Definition 2: a state *is* the set of possible actions. Hoare's refusal to distinguish ontologically between "active" agents and "passive" objects anticipates our Principle 1.

Milner's *Calculus of Communicating Systems* (CCS) [Milner, 1989] introduces *observation* as the fundamental mode of interaction. Two processes are equivalent if no observer can distinguish them through interaction—a criterion that resonates with our treatment of observable reality as the ultimate selector (Principle 11). Milner's later π-calculus [Milner, 1999] extends this to model *mobility*: the dynamic reconfiguration of communication topology. This is directly relevant to our discussion of architectural decisions as partitioning of action possibilities (Principle 8), since the π-calculus models changing connectivity as itself an action.

Distinction from our project. While process algebras provide *mathematical* frameworks for specifying and verifying concurrent systems, our framework addresses a different question: the *epistemological* status of software artifacts across the development lifecycle. Process algebras operate at what we call Level 3 (Conceptions), they are formal languages for expressing action possibilities. Our contribution is the vertical integration: explaining how Level 3 formalisms relate to Level 2 perceptions held by stakeholders, Level 1 observations of running systems, and the evolutionary dynamics by which decisions at all levels are selected by reality.

Process algebras could serve as a formal substrate for expressing the ground model and subsequent refinements within our framework. The semantic equivalences of process algebra (trace equivalence, bisimulation) provide rigorous criteria for when two specifications denote "the same" action possibilities—a question our framework raises but does not itself resolve mathematically. Future work might explore ASMs [Börger, 1999] or process-algebraic notations as the formal language for action-centric specifications.

7.2 Foundational Software Engineering Ontologies

The dominant tradition in software engineering ontology derives from Mario Bunge's *substance ontology* [Bunge, 1977], which holds that the world fundamentally consists of *things* (substantial individuals) possessing *properties*. This "thing-centric" view has been systematically applied to information systems through the **Bunge-Wand-Weber (BWW) models** [Wand & Weber, 1990].

In the BWW framework, an information system represents a domain composed of things, their properties, and the states those things occupy. State changes occur through *events*, but events are secondary, they are what happens *to* things. This ontological priority is reversed in our framework: actions (events, processes) are primary, and things are constructed as invariants under cognitive action (Definition 3).

The BWW model has proven valuable for evaluating modeling grammars and identifying representational deficiencies in notations like ER diagrams and UML. However, its thing-centric foundation creates systematic difficulties for representing phenomena that resist "thingification":

- **Processes and workflows** are awkwardly modeled as things with temporal properties, rather than as the primary reality they constitute in business domains
- **Services and capabilities** (what a system *can do*) are subordinated to the objects that provide them
- **Intent and rationale** have no natural place in a thing-property ontology

Guizzardi's **Unified Foundational Ontology (UFO)** [Guizzardi, 2005] offers a more sophisticated metaphysics, distinguishing *Endurants* (entities that persist through time while remaining wholly present at each moment) from *Perdurants* (events and processes that unfold through time). UFO thus acknowledges both things and events as ontological categories.

Our framework can be understood as *collapsing* this distinction: what UFO calls Endurants are, in our view, constructed invariants arising from patterns of Perdurants. The "user account" is not an Endurant that *participates in* events; it is a cognitive construction that *emerges from* the regularities stakeholders observe across authentication, authorization, and profile-management actions. This is a substantive theoretical claim, not merely a notational preference.

Implications of the disagreement. If we are correct that objects are constructed from action patterns, then software modeling should begin with action analysis and derive object models, rather than the reverse. The persistent difficulty of "finding the right objects" in object-oriented analysis may stem from treating as primitive what is actually derivative. Domain-Driven Design's emphasis on "ubiquitous language" and behavioral modeling [Evans, 2004] implicitly recognizes this, though without the ontological foundation we provide.

Conversely, if the BWW/UFO view is correct, our framework conflates epistemology (how we come to know objects) with ontology (what objects are). We acknowledge this tension. Our defense is pragmatic: for the purposes of software development, constructing systems that satisfy human intent, the epistemological question is primary. We need not resolve whether objects "really exist" independently of action; we need only recognize that *for software development purposes*, treating actions as primary yields better traceability from intent to implementation.

7.3 Methodological and Requirements Approaches

Goal-oriented requirements engineering [van Lamsweerde, 2001] shares our emphasis on intent but typically models goals as objects in a goal hierarchy rather than as constraints on action possibilities. The KAOS framework's operationalization of goals into operations comes closest to our view, though it retains an object-centric domain model alongside the goal model.

Behavioral specifications such as TLA+ (Lamport), Alloy (Jackson), and Abstract State Machines [Gurevich, 1995, Börger, 1999] provide formal languages for specifying action possibilities. ASMs are particularly relevant: they define computation as guarded updates to abstract states, which aligns with our "actions changing action possibilities." Börger's methodology of *ground model* construction and stepwise refinement [Börger, 1999] directly informs our Section 2's treatment of the ground model problem. Our contribution is situating ASM methodology within a broader epistemological framework that explains *why* the ground model occupies its privileged position.

Domain-driven design [Evans, 2004] emphasizes constructed models and ubiquitous language but treats domain models as representing an external reality rather than as invariants under cognitive action. Evans's insight that different bounded contexts may model "the same" domain differently is compatible with our Proposition on Object Relativity, though DDD lacks the theoretical apparatus to explain *why* this relativity arises.

Agile methods implicitly recognize the formality continuum by preferring conversation over documentation, but lack a theoretical foundation for determining *when* higher formality is appropriate. Our Principle 7 (Context-Dependent Optimality) provides this foundation.

8 Conclusion

We have presented an action-centric ontology for software development grounded in Turchin's cybernetic philosophy and Heylighen's theory of formality. The framework unifies traditionally separate activities under a coherent world view:

- Actions are ontologically primary; states are action possibilities
- Objects are constructed invariants under cognitive actions
- Artifacts are decisions about action possibilities at varying formality
- Validation is selection by observable reality
- Progress occurs through metasystem transitions

By positioning this framework against both the process-algebraic tradition (which provides compatible mathematical foundations) and the dominant thing-centric ontologies in software engineering (BWW, UFO), we have clarified its distinctive contribution: vertical integration across abstraction levels, from stakeholder perceptions through formal specifications to observable system behavior.

Practical implications include new approaches to requirements specification, architectural description, testing strategy, and tool design. Future work will develop formal languages for action-centric specification, potentially drawing on process algebras or Abstract State Machines, and empirically evaluate the framework in industrial practice.

References

Börger, E. (1999). High Level System Design and Analysis using Abstract State Machines. In *Current Trends in Applied Formal Methods (FM-Trends 98)*, LNCS Vol. 1641, pp. 1–43. Springer-Verlag.

Bunge, M. (1977). *Treatise on Basic Philosophy: Ontology I: The Furniture of the World.* D. Reidel Publishing Company, Dordrecht.

Campbell, D.T. (1974). Evolutionary Epistemology. In Schilpp, P.A. (ed.), *The Philosophy of Karl Popper*, pp. 413–463. Open Court Publishing.

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

Guizzardi, G. (2005). *Ontological Foundations for Structural Conceptual Models.* Ph.D. thesis, University of Twente. CTIT Ph.D. Thesis Series No. 05-74.

Gurevich, Y. (1995). Evolving Algebras 1993: Lipari Guide. In Börger, E. (ed.), *Specification and Validation Methods*, pp. 9–36. Oxford University Press.

Heylighen, F. (1999). Advantages and limitations of formal expression. *Foundations of Science*, 4(1):25–56.

Heylighen, F. (2000). Foundations and methodology for an evolutionary world view: A review of the Principia Cybernetica Project. *Foundations of Science*, 4(3):457–490.

Hoare, C.A.R. (1985). *Communicating Sequential Processes.* Prentice Hall International.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Milner, R. (1989). *Communication and Concurrency.* Prentice Hall.

Milner, R. (1999). *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press.

Turchin, V. (1977). *The Phenomenon of Science.* Columbia University Press, New York.

Turchin, V. (1993). The cybernetic ontology of action. *Kybernetes*, 22(2):10–30.

van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proc. 5th IEEE Int. Symp. Requirements Engineering*, pp. 249–262.

Wand, Y. and Weber, R. (1990). An ontological model of an information system. *IEEE Trans. Software Engineering*, 16(11):1282–1292.